VI.04.

Общий обзор занятия.

Продолжая разговор о структурах данных и бинарном дереве поиска, говорим о сбалансированных деревьях поиска, точнее об одном варианте сбалансированного дерева поиска - об АВЛ-дереве. Собственно, эта структура - и есть вопрос занятия.

Ничего другого, пожалуй и не впишется сюда. Но обязательно ещё раз потоптаться на содержательной части интерфейсов: что это такое, зачем, что означает и где и как применяется, те же вопросы про композицию/встраивание интерфейсов. И опять как можно более содержательнее, по возможности неформально. Более тщательный и формальный разговор об интерфейсах - в ближайшем будущем, а в этот раз - фактически последняя возможность как-то с ними сжиться.

Формально говоря, мы говорим сегодня о некотором навороте над двоичным деревом поиска, о его балансировке. В связи с этим довольно логично, что

package order - пакет с общими константами, типами, переменными и т.д., в т.ч. и интерфейсами:

совершенно не изменится. Чисто для удобства чтения поместим здесь его текст:

```
package order

type Ordered interface {
    Before(b Ordered) bool
}

type Visual interface {
    Show() string
}

type Key interface {
    Ordered
    Visual
}
```

```
const (
    PreOrder         = iota // NLR
    InOrder                 // LNR = ascending order
    PostOrder               // LRN
    ReversePreOrder         // NRL
    ReverseInOrder          // RNL = descending order
    ReversePostOrder        // RLN
)

var ImageWidth int

func init() {
    ImageWidth = 5
}
```

А потом поедем дальше. А дальше говорим про сбалансированные деревья поиска вообще (немножко) и, в частности, про АВЛ-дерево - а вот про него подробно и вдумчиво.

Сбалансированное дерево поиска: АВЛ-дерево.

Проблема состоит в следующем - при неблагоприятных обстоятельствах бинарное дерево поиска может оказаться сильно перекошенным, из-за этого операции вставки/удаления на нём могут исполняться долго. В самом крайнем случае, если элементы, которые мы вставляем в дерево поиска, приходят в порядке возрастания (или убывания), то дерево состоит из одной очень длинной ветки. Оказывается, есть средство борьбы с таким поведением дерева поиска. Надо просто не давать ему сильно перекашиваться и расти очень далеко (в смысле глубоко). А как мы это можем сделать? Ведь каждый элемент при вставке фактически сам выбирает своё место, у нас нет способов повлиять на выбор его места в дереве поиска. Нет - и не надо. Будем просто преобразовывать дерево так, чтобы элементы остались те же, дерево оставалось деревом поиска, но форма дерева изменилась в лучшую сторону, чтобы

оно было не таким перекособоченным. Называется это коротко “сбалансировать дерево”, более точно, “сбалансировать по высоте”. Существует немало способов балансировки дерева поиска, например, 2-3-деревья или красно-чёрные деревья. Мы рассмотрим только один способ балансировки дерева поиска - АВЛ-дерево. Основные операции на нём видны очень отчётливо, да и результаты оно даёт вполне себе доброкачественные (в смысле эффективности). Вот им и займёмся.

Понятие сбалансированного дерева поиска.

Основные операции на деревьях поиска - вставка, удаление и поиск элемента - проходят по дереву сверху вниз и могут потребовать столько операций, какова высота дерева. Точнее говоря, не столько, а пропорционально высоте дерева - мало ли, сколько операций и какую обработку мы может учинять в каждой вершине по пути. При идеальном раскладе двоичное дерево высоты h содержит 2^h-1 узлов, т.е. высота дерева с N узлами может быть равна примерно $\log_2 N$. Мы об этом говорили немного в связи с разговорами о бинарной куче. Но это в идеальном случае. А может оказаться совсем наоборот - если, например, мы будем добавлять ключи в дерево в порядке возрастания или убывания, то дерево вытянется в одну сторону, фактически мы получим связный список. И тогда высота дерева с N узлами будет равна N. А это, мы видели, неизмеримо хуже. Т.е. в худшем случае дерево поиска может работать очень медленно. Да, конечно, если данные приходят и уходят рандомизированно, то очень маловероятно (практически невероятно), что мы получим очень сильно перекошенное дерево, но на практике рандомизация (случайное перетасовывание) данных очень часто недопустима, а варианты, когда входные данные приходят в почти отсортированном порядке не так уж редки. Так что опасность есть, и немалая. Что же делать? Ну, раз мы не можем управлять потоком входных данных, значит нам надо искать ресурсы внутри, в самом дереве в данном случае. Надо как-то сделать так, чтобы дерево не росло очень сильно в высоту (глубину) при любых данных. Но мы же видели, что при вставке элемент совершенно однозначно занимает свое место. Выход есть: вставляем терпеливо этот элемент, куда ему полагается, а потом берём и перетряхиваем как-то получающееся дерево с целью не дать ему сильно расти в высоту, оставаясь при этом деревом поиска. Такая операция называется, вполне естественно, балансировкой или балансировкой по высоте. Ну так что, берём новый элемент, добавляем его и из имеющихся элементов конструируем идеально сбалансированное дерево? Это было бы замечательно, если бы мы умели делать эту операцию быстро. А не умеем... Да это и невозможно (без объяснений, придётся

верить на слово). Но мы можем придумать какие-нибудь варианты дерева поиска, которые растут не очень быстро с ростом количества элементов в нём, примерно с такой же скоростью, с какой растёт идеально сбалансированное дерево, ну немного быстрее, раза в полтора-два, и при этом перетряхивание дерева при добавлении и удалении элементов исполнялось бы достаточно быстро. Такие деревья называются сбалансированными деревьями поиска, или, точнее говоря, сбалансированными по высоте.

При этом важно (причём надо бы проговорить это явно), отметить такй момент. При перетряхиваниях дерева поиска может нарушаться условие “значение ключа в левом сыне строго меньше значения ключа в родителе”, они могут оказаться равными. При удалении элементов это может привести к некорректностям, об этом уже говорили раньше, обсуждая удаление элемента из произвольного дерева поиска (этот момент хорошо бы проговрить по-быстрому при обсуждении удаления из произвольного дерева поиска). В произвольном дереве мы заменяли удаляемый узел на узел с наименьшим значением ключа в правом поддереве удаляемого узла, а не на узел с наибольшим значением ключа в левом поддереве именно для того, чтобы избежать таких неприятностей. Но в сбалансированных деревьях условие “ключ слева строго меньше” в принципе невыполнимо. Да, конечно, если вставлять различные ключи в возрастающем порядке, то дерево вытянется в линию, но его можно как-то изменить, чтобы оно стало пониже, не таким вытянутым, но что будет, если, например, дерево состоит из нескольких одинаковых ключей? Тогда по нашим правилам “меньшие налево, бОльшие или равные направо” нет другой возможности построить дерево поиска из однаковых ключей, кроме как вытягивание дерева поиска в линию вправо. Можно, конечно, вовсе отказаться от повторяющихся ключей, утешаясь соображениями типа “ну, оно же дерево поиска, так чего втыкать в дерево уже имеющийся элемент; если он там уже есть, так мы его найдём успешно...”. Но это так, утешения, можно придумать кучу ситуаций, когда необходимо держать в дереве элементы с одинаковыми ключами поиска, хотя сами элементы будут различаться. Мы не пойдём таким путём, мы разрешим держать в сбалансированном дереве поиска элементы с равными ключами. При этом вставка будет происходить, как и раньше, по правилу “меньшие налево, больше или равные - направо”. Но как уже говорилось выше, при балансировке это условие может (и будет) нарушаться. Но тогда появляются проблемы с удалением элемента, которые довольно подробно обсуждались выше - в разговоре о произвольном бинарном дереве поиска. Вот здесь мы и изменим код удаления элемента. Давайте сразу здесь её и опишем, благо идея совсем простая. В чём был источник проблем? Вот мы хотим удалить узел. Ищем самый левый элемент в его правом поддереве и запоминаем его значение, скажем minkey. А потом мы ставили

этот узел на место удаляемого и удаляли из правого поддерева удаляемого узла узел, со значением ключа, равным minkey. И вот тут-то нас и ждёт засада - может оказаться, что ключ minkey встретится по дороге к самому левому узлу (узлу, у которого нет левого сына). Так давайте просто при поиске узла для удаления не останавливаться на полдороги - если у ключа есть левый сын с таким же значением ключа, то продолжим движение. Изменения в коде минимальные.

Мы рассмотрим один из вариантов сбалансированного дерева поиска - АВЛ-дерево. Вариантов много, перечислять названия смысла нет. Дадим определение АВЛ-дерева и эффективно реализуем основные операции на нём. А вот доказывать, что АВЛ-дерево действительно не растёт слишком сильно, мы не будем, просто скажем, что высота АВЛ-дерева с N вершинами не превосходит $1.45 \cdot \log_2 N$, что совсем неплохо.

АВЛ-дерево. Определение.

АВЛ-дерево — это двоичное дерево поиска, обладающее одной особенностью: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Эта особенность гарантирует, что дерево высоты h содержит не менее $(1+\sqrt{5})^h$ узлов. Т.е. высота дерева растёт не очень быстро с ростом количества узлов. Но об этом абзацем выше. И без доказательств. Там несложно, всё в рамках школьной математики, но не будем отвлекаться.
Замечание. Говоря совсем точно, если обозначить $N(h)$ минимальное возможное количество узлов в АВЛ-дереве высоты h, то $N(1) = 1$, $N(2) = 2$, и $N(h) = N(h-1) + N(h-2) + 1$ для $h>=3$. Кто хочет, может это доказать.

Почему такое странное название? Структура названа в честь авторов, придумали её аж в 1962 году Г.М. Адельсон-Вельский и Е.А. Ландис.

АВЛ-дерево - основные операции и их реализация.

Но давайте уже перейдём к делу - к операциям с АВЛ-деревом. Поскольку АВЛ-дерево является деревом поиска, то операции, не связанные с изменением формы дерева - поиск элемента в дереве, обход дерева, визуализации дерева - полностью сохраняются. Остаются, естественно, две операции - добавление и удаление элемента. О них и поговорим.

Но сначала объявим типы. Тип дерева остаётся без изменений, а вот, поскольку в определение АВЛ-дерева входят высоты поддеревьев, добавим к каждому узлу высоту

поддерева, корнем которого он является:

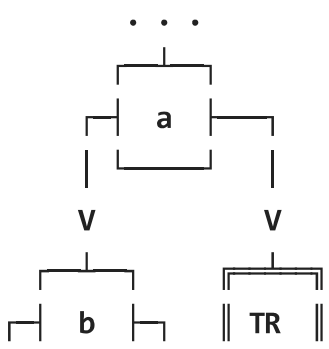
```
type (
    Node struct {
        Value  order.Key
        Lson   AVLtree
        Rson   AVLtree
        height int
    }
    AVLtree struct {
        root *Node
    }
)
```

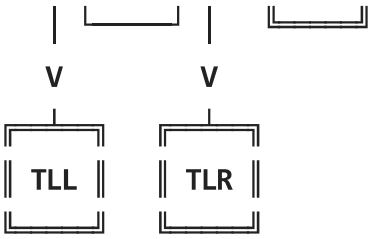
АВЛ-дерево. Балансировка вершин при операциях вставки и удаления.

Операции вставки/удаления элемента в АВЛ-дерево выполняются точно также, как и в произвольном дереве поиска. Однако после того, как изменится поддерево с корнем в данной вершине, требуется исполнить балансировку этого поддерева.

Итак, ситуация следующая: мы находимся в вершине, содержащей ключ `a`; оба поддерева вершины `a` сбалансированы - удовлетворяют условиям АВЛ-дерева. Если высоты этих поддеревьев отличаются не более, чем на 1, то ничего делать не надо. А вот если высоты отличаются на 2, то преобразуем поддерево с вершиной `a`. Сразу заметим, что отличаться больше, чем на 2, высоты поддеревьев не могут, мы ведь изменяли одно из них только на один элемент - добавляли элемент или удаляли его.

Допустим высота левого поддерева вершины `a` на 2 больше высоты правого поддерева вершины `a`. Посмотрим на рисунок:

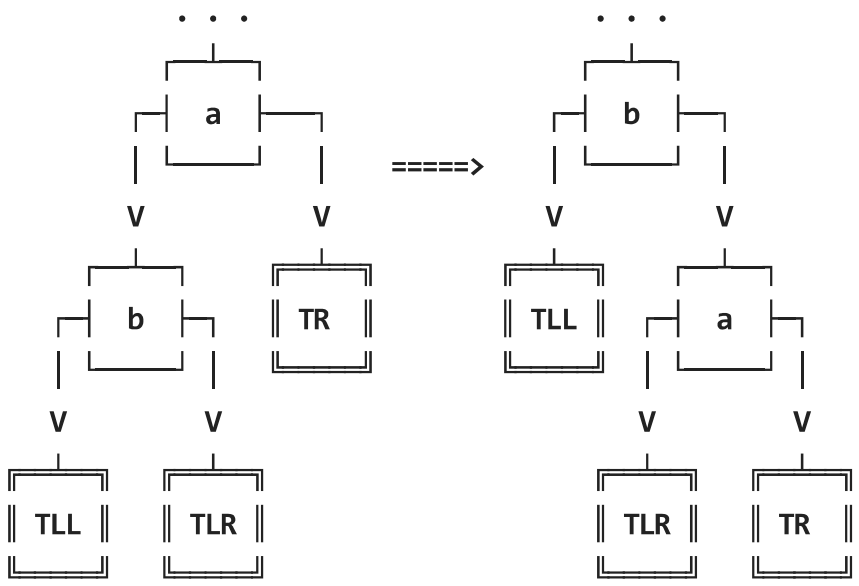




Будем обозначать h(a) высоту поддерева с корнем в вершине `a`, а высоту поддерева T обозначим h(T). Пусть h(TR) = H, тогда h(b) = H+2. Возможны три варианта:

1. h(TLL) = H+1, h(TLR) = H+1
2. h(TLL) = H+1, h(TLR) = H
3. h(TLL) = H, h(TLR) = H+1

Рассмотрим их все.
1. h(TLL) = H+1, h(TLR) = H+1.
Выполним преобразование, которое называется левый поворот, и изображено на рисунке:



Легко видеть, что после преобразования дерево остаётся деревом поиска. Корень поддерева теперь находится в вершине b. Высота левого поддерева вершины a равна H+1, высота правого - H+2. Напомню, что h(TLR) = H+1, h(TR) = H. Так что и поддерево с корнем a, и дерево с корнем b являются АВЛ-деревьями.
2. h(TLL) = H+1, h(TLR) = H.
В этом случае выполняем точно такое же преобразование. Вся разница только в том, что после преобразования h(a) = H+1, h(TLR) = h(TR) = H.

Понятно, что если высота правого поддерева вершины `a` на 2 больше высоты левого поддерева вершины `a`, то ситуация разрешается совершенно симметрично - правым поворотом.

Вот код обоих поворотов (понятно, что можно объединить эти повороты в одну функцию, но так яснее всё видно, пусть будут две функции):

```
// left rotate a tree, and update tree's height as well as height of all subtrees
func (t *AVLtree) singleRotateLeft() {
    var left AVLtree
    if !(*t).Empty() {
        // turn left
        left = (*(*t).root).lson
        (*(*t).root).lson = (*left.root).rson
        (*left.root).rson = *t

        //update height
        (*t).updateHeight()
        left.updateHeight()

        *t = left
    }
}

// right rotate a tree, and update tree's height as well as height of all subtrees
func (t *AVLtree) singleRotateRight() {
    var right AVLtree
    if !(*t).Empty() {
        // turn right
        right = (*(*t).root).rson
        (*(*t).root).rson = (*right.root).lson
        (*right.root).lson = *t

        //update height
        (*t).updateHeight()
        right.updateHeight()

        *t = right
```

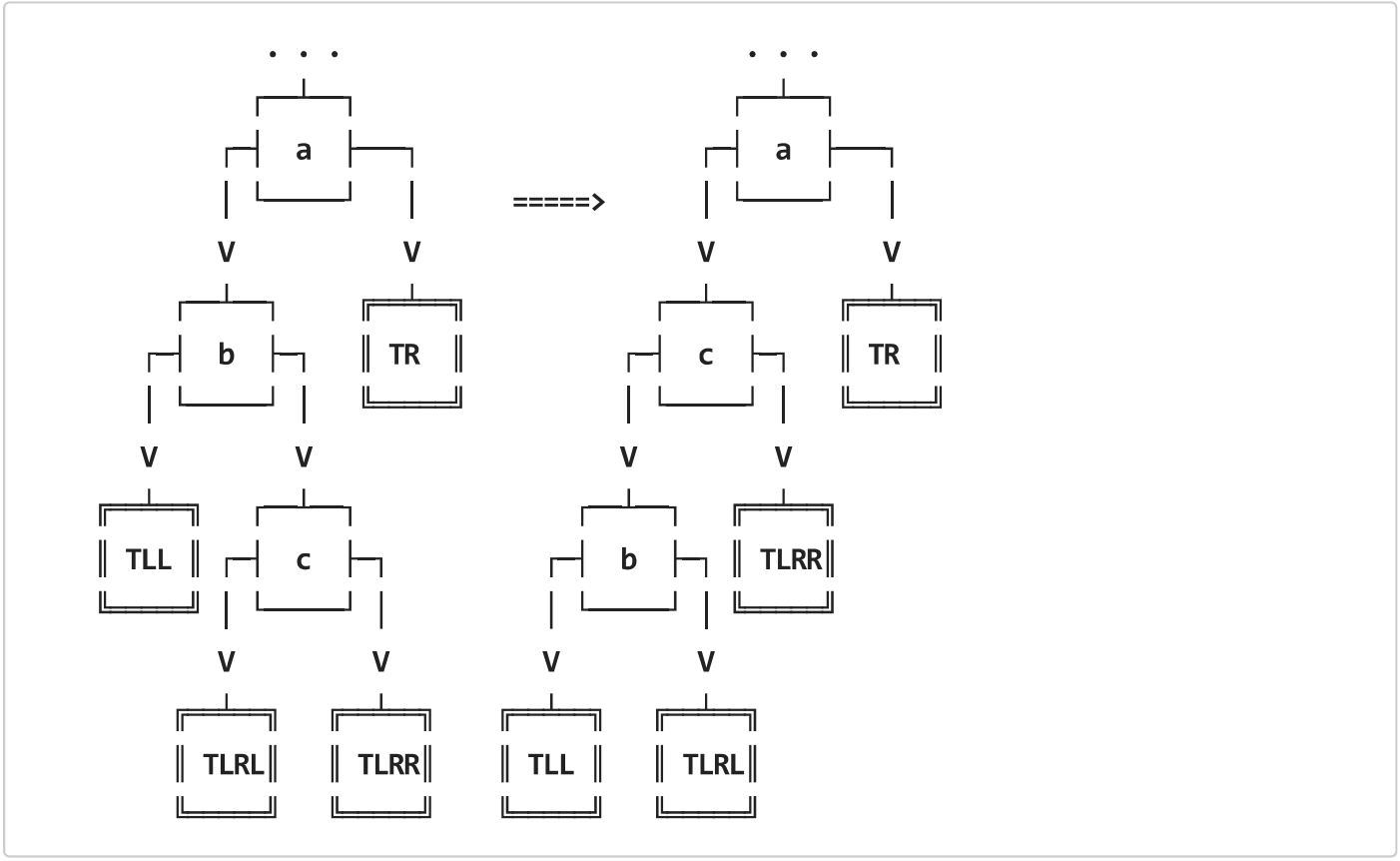
```
    }
}
```

Да, и вот код, перевычисляющий высоту дерева при поворотах:

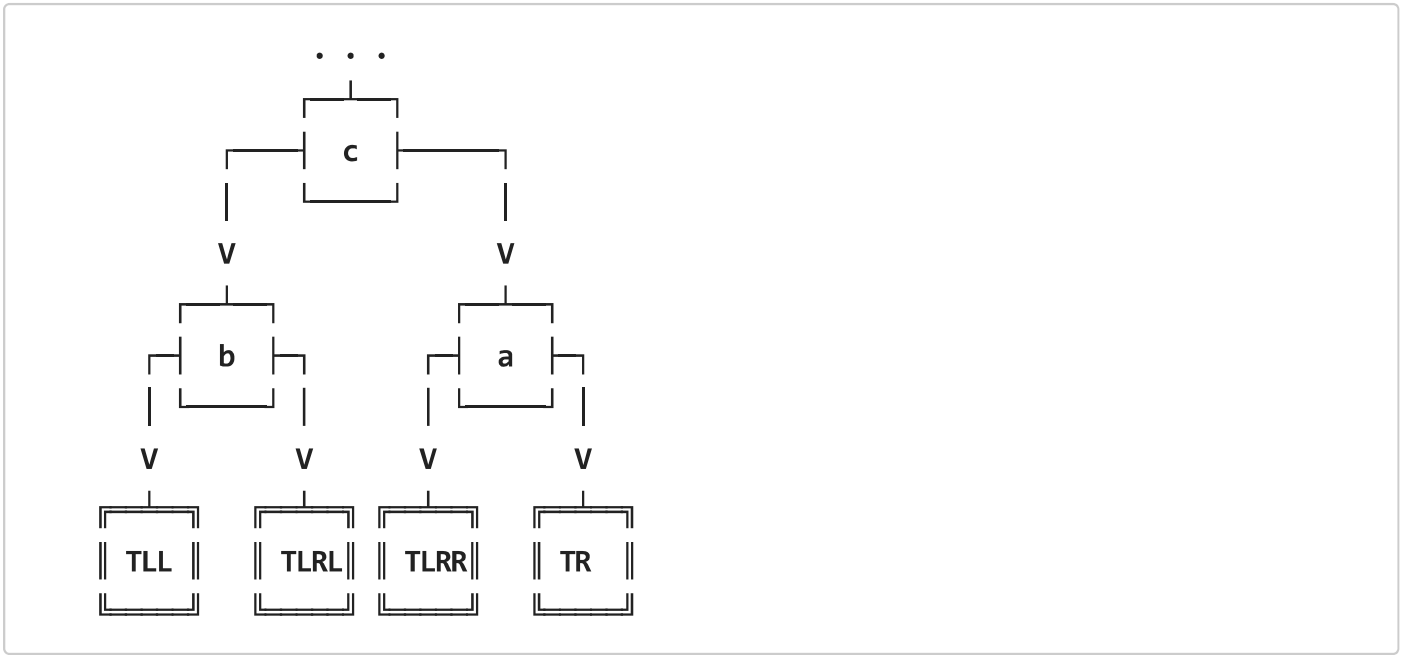
```
//recalculate the height of the node
func (t *AVLtree) updateHeight() {
    if (*t).Empty() {
        return
    }
    if (*(*t).root).lson.treeHeight() > (*(*t).root).rson.treeHeight() {
        (*(*t).root).height = (*(*t).root).lson.treeHeight() + 1
    } else {
        (*(*t).root).height = (*(*t).root).rson.treeHeight() + 1
    }
}

//return the height of the node
func (t AVLtree) treeHeight() int {
    if t.Empty() {
        return 0
    } else {
        lh, rh := (*t.root).lson.treeHeight(), (*t.root).rson.treeHeight()
        if rh > lh {
            return rh + 1
        } else {
            return lh + 1
        }
    }
}
```

Остался ещё один случай:
3 случай. (TLL) = H, h(TLR) = H+1.
А вот здесь левый поворот не срабатывает. Если его выполнить, то правое поддерево вершины b будет иметь высоту H+2, а левое - только H. Поэтому сначала выполним правый поворот поддерева с вершиной b. Повторим, правый поворот выполняется точно также, как и левый, только в противоположную сторону. В результате получим следующее:



Заметим, что h(c) = h(TLR) = H+1, а из этого следует, что высоты поддеревьев TLRL и TLRR равны H или H-1. Напомню, что h(TLL) = H = h(TR).
Выполним теперь левый поворот поддерева с вершиной a и придём к окончательному результату:



Высоты деревьев TLL и TLRL отличаются не более, чем на 1, также как и высоты деревьев TLRR и TR, при этом высоты дервьев TLRL и TLRR равны H или H-1.

Следовательно h(b) = h(a) = H+1, и мы получили АВЛ-дерево.

Абсолютно точно также, только с заменой левых поворотов на правые, а правых на левые, мы будем действовать, если высота левого поддерева вершины a на 2 меньше высоты правого поддерева вершины a.

Остаётся только обновить значения высот в узлах a, b и c.

Код таких “двойных” поворотов вот:

```
// v = subtree root, vl = v's left child, vlr = vl's right child
// right rotate vl & vlr, left rotate v & v's left child
// return a new tree
func (t *AVLtree) doubleRotateLeftRight() {
    //right rotatel between vl & vlr
    (*(*t).root).lson.singleRotateRight()

    //left rotate between v and his left child
    (*t).singleRotateLeft()
}

// v = subtree root, vr = vr's right child, vrl = vr's left child
// left rotate vr & vrl, right rotate v & v's right child
// return a new tree
func (t *AVLtree) doubleRotateRightLeft() {
    //left rotatel between vr & vrl
    (*(*t).root).rson.singleRotateLeft()

    //right rotate between v and his left child
    (*t).singleRotateRight()
}
```

Для полноты картины приведём коды операций вставки и удаления ключа в АВЛ-дереве:

```
//insert an x to AVLtree
func (t *AVLtree) Insert(x order.Key) {
    if (*t).Empty() {
        //new(Node)
```

```
        (*t).root = &Node{
            Value:  x,
            height: 1,
        }
    } else if x.Before((*(*t).root).Value) {
        (*(*t).root).lson.Insert(x)
        if (*(*t).root).lson.treeHeight()-(*(*t).root).rson.treeHeight() == 2 {
            if x.Before((*(*(*t).root).lson.root).Value) { //left left
                (*t).singleRotateLeft()
            } else { //left right
                (*t).doubleRotateLeftRight()
            }
        }
    } else /* !x.Before((*(*t).root).Value) */ {
        (*t).root.rson.Insert(x)
        if (*(*t).root).rson.treeHeight()-(*(*t).root).lson.treeHeight() == 2 {
            if !x.Before((*(*(*t).root).rson.root).Value) {
                (*t).singleRotateRight()
            } else {
                (*t).doubleRotateRightLeft()
            }
        }
    }
    (*t).updateHeight()
}

//delete an x in AVLtree
func (t *AVLtree) Delete(x order.Ordered) {
    // Здесь мы никак не используем и не будем использовать в дальнейшем x.Show(), так
 что
    // можно считать, что мы удаляем ключ именно интерфейса order.Ordered, а не order.
Key
    if !(*t).Empty() {
        if x.Before((*(*t).root).Value) || (!(*(*t).root).lson.Empty() && !(*(*(*t).ro
ot).lson.root).Value.Before(x)) { // <= - главное отличие удаления
            (*(*t).root).lson.Delete(x)
            if (*(*t).root).rson.treeHeight()-(*(*t).root).lson.treeHeight() == 2 {
                if (*(*(*t).root).rson.root).lson.treeHeight() <= (*(*(*t).root).rson.
```

```
root).rson.treeHeight() {
                    (*t).singleRotateRight()
                } else {
                    (*t).doubleRotateRightLeft()
                }
            }
        } else if (*(*t).root).Value.Before(x) {
            (*(*t).root).rson.Delete(x)
            if (*(*t).root).lson.treeHeight()-(*(*t).root).rson.treeHeight() == 2 {
                if (*(*(*t).root).lson.root).rson.treeHeight() <= (*(*(*t).root).lson.
root).lson.treeHeight() {
                    (*t).singleRotateLeft()
                } else {
                    (*t).doubleRotateLeftRight()
                }
            }
        } else if (*(*t).root).lson.Empty() {
            *t = (*(*t).root).rson
        } else if (*(*t).root).rson.Empty() {
            *t = (*(*t).root).lson
        } else
        /* !leftsubtree.Empty() && !rightsubtree.Empty() */ {
            min := (*(*t).root).rson.leftmost()
            (*(*t).root).rson.Delete((*min).Value)
            (*min).lson, (*min).rson = (*(*t).root).lson, (*(*t).root).rson
            (*t).root = min
            if (*(*t).root).lson.treeHeight()-(*(*t).root).rson.treeHeight() == 2 {
                if (*(*(*t).root).lson.root).rson.treeHeight() <= (*(*(*t).root).lson.
root).lson.treeHeight() {
                    (*t).singleRotateLeft()
                } else {
                    (*t).doubleRotateLeftRight()
                }
            }
        }
    }
    (*t).updateHeight()
}
```

```
//find leftmost min elem in the subtree
func (t AVLtree) leftmost() *Node {
    if t.Empty() {
        return nil
    }
    for !(*t.root).lson.Empty() {
        t = (*t.root).lson
    }
    return t.root
}
```

При этом хочется обратить внимание на следующее: метод вставки вставляет в дерево ключ типа `order.Key`

```
func (t *AVLtree) Insert(x order.Key) {
```

а метод удаления удаляет ключ типа `order.Ordered`

```
func (t *AVLtree) Delete(x order.Ordered) {
```

Вот тут как раз стоит поговорить о расщеплениях и экономии интерфейсов. Ну, то, что мы вставляем в АВЛ-дерево `order.Key`, понятно - раз есть метод визуализации дерева, который требует визуализовывать значения в узлах, так если вставить ключ типа `order.Ordered`, то случится облом - `order.Ordered` не имеет метода Show(), ответственного за внешний вид ключа. А вот удаляя ключ, нам нужно только искать его, сравнивая с другими ключами, визуализовать его при этом не нужно и, заметим, и не будет больше нужно никогда, раз уж мы его удаляем из дерева, так что тут вполне достаточно указывать ключ типа `order.Ordered`. Кстати, всё сказанное про метод удаления относится и к методу поиска ключа

```
func (t AVLtree) Find(n order.Ordered) (node *Node, ok bool)
```

Собственно, всё это было уже в прошлый раз - всё это относится и к обычному бинарному дереву поиска, но лишний раз повторить (даже если в прошлый раз удалось про это сказать явно) - в данном случае точно не помешает.

Ну, и, поскольку операции поиска ключа в АВЛ-дереве, обхода дерева и визуализации

дерева не связаны с изменением дерева, они для АВЛ-дерева ничем не отличаются от соответствующих операций для произвольного бинарного дерева поиска. Соответствующий код приводится ниже, чисто для полноты картины.

```
func (t AVLtree) Find(n order.Ordered) (node *Node, ok bool) {
    // Здесь мы никак не используем x.Show(), так что для поиска нам
    //  достаточно иметь ключ интерфейса order.Ordered, а не order.Key
    switch {
    case t.Empty():
        return nil, false
    case n.Before((*t.root).Value):
        //    n before (*t.Root).Value
        return (*t.root).Lson.Find(n)
    case (*t.root).Value.Before(n):
        //    n after (*t.Root).Value
        return (*t.root).Rson.Find(n)
    default:
        // n is equivalent to (*t.Root).Value
        return t.root, true
    }
}

func (t AVLtree) Traversal(dir int, f func(x order.Key)) {
    if t.Empty() {
        return
    }
    switch dir {
    case order.PreOrder: // NLR
        f((*t.root).Value)
        (*t.root).Lson.Traversal(dir, f)
        (*t.root).Rson.Traversal(dir, f)
    case order.InOrder: // LNR
        (*t.root).Lson.Traversal(dir, f)
        f((*t.root).Value)
        (*t.root).Rson.Traversal(dir, f)
    case order.PostOrder: // LRN
        (*t.root).Lson.Traversal(dir, f)
        (*t.root).Rson.Traversal(dir, f)
```

```
            f((*t.root).Value)
        case order.ReversePreOrder: // NRL
            f((*t.root).Value)
            (*t.root).Rson.Traversal(dir, f)
            (*t.root).Lson.Traversal(dir, f)
        case order.ReverseInOrder: // RNL
            (*t.root).Rson.Traversal(dir, f)
            f((*t.root).Value)
            (*t.root).Lson.Traversal(dir, f)
        case order.ReversePostOrder: // RLN
            (*t.root).Rson.Traversal(dir, f)
            (*t.root).Lson.Traversal(dir, f)
            f((*t.root).Value)
        }
}


func (t AVLtree) Diagram() []string {
    if t.Empty() {
        return []string{""}
    }
    sL := (*t.root).Lson.Diagram()
    sR := (*t.root).Rson.Diagram()
    spaceL := strings.Repeat(" ", stringLen(sL[0]))
    for len(sL) < len(sR) {
        sL = append(sL, spaceL)
    }
    spaceR := strings.Repeat(" ", stringLen(sR[0]))
    for len(sR) < len(sL) {
        sR = append(sR, spaceR)
    }
    r := []rune(sL[0])
    ch := ' '
    for i := range r {
        if r[i] == '|' {
            r[i] = '┌'
            ch = '—'
        } else {
```

```
            r[i] = ch
        }
    }
    sL[0] = string(r)
    r = []rune(sR[0])
    ch = '─'
    for i := range r {
        if r[i] == '│' {
            r[i] = '┐'
            ch = ' '
        } else {
            r[i] = ch
        }
    }
    sR[0] = string(r)

    s := []string{spaceL + fmt.Sprintf("%*s", order.ImageWidth, "│") + spaceR}
    s = append(s, sL[0]+(*t.root).Value.Show()+sR[0])
    for i := 1; i < len(sL); i++ {
        s = append(s, sL[i]+strings.Repeat(" ", order.ImageWidth)+sR[i])
    }
    return s
}

func stringLen(s string) int {
    return len([]rune(s))
}
```

Задания для самостоятельной работы

Каких-то специальных задач тут не особо видно, дерево поиска встречается где угодно, инструмент важный и полезный, но это именно инструмент. Тем не менее какие-то задания сделать было бы неплохо.

1. Поработать с деревом поиска и, особенно, с АВЛ-деревом вручную: повставлять/поудалять элементы, последить, как работают алгоритмы вставки и удаления, как работают повороты, разобраться с ними, понять их.

2. Собрать из ключей дерево поиска - это уже фактически сортировка. Т.е. алгоритм простой: загнать числа в дерево поиска и, обойдя дерево поиска, собрать числа в возрастающем порядке. Задание: написать такую сортировку на бинарном дереве поиска и на АВЛ-дереве. Сравнить их по скорости на случайном массиве, на массиве, почти упорядоченном в возрастающем порядке, на массиве, почти упорядоченном в убывающем порядке. Сравнить их по скорости с QuickSort, MergeSort и HeapSort.

Теоретический материал про АВЛ-деревья

в принципе изложен в этом тексте, но в материалах к занятию выложен фрагмент книги Окулов С.М. Абстрактные типы данных. - М.: Лаборатория знаний, 2020. с.178-188., где изложен анализ скорости роста АВЛ-дерева, расписаны операции с деревом, картинки там покрасивее, так что есть смысл дать детям.